Operations Research Methods

Genetic Algorithm

Anne Barnasconi: 2053988

Dico de Gier: 2058017

May 26, 2023

1 Algorithmic Choices

1.1 Chromosome

First of all, we represent a chromosome by the use of a cell. If our initial matrix is $M \in \mathbb{R}^{m \times n}$ and we are given a maximum of K_{max} elements, then a chromosome is a 1 by K_{max} cell. Now say a solution i has $K(i) \leq K_{max}$ elements, then each column index smaller or equal $K(i)$ contains an $m \times n$ matrix. This $m \times n$ matrix is the row-concave matrix, which is elementally-wise multiplied with the weight of that matrix. All column indices larger than $K(i)$ contain an empty matrix. We chose this approach, because a cell provides a clear overview and quick access to the different matrices. Also, by multiplying the matrices element-wise with their weights, we do not have to store another data structure with the weights, since we can simply extract them from the matrices by using a cell function.

1.2 Initial population

We create an initial population by randomly generating the number of matrix elements per solution, the matrices themselves and the weights corresponding to a matrix. First, we created a function called `initial_solutions`. Our parameter P, which is the population size, determines how many initial solutions we create. Firstly, the function generates a random vector of size P, which will determine how many matrix elements each solution will have on the right hand side. We chose to do this because we found that some initial solutions with a low number of matrices had relatively good objective values compared to initial solutions with a high number of matrices, and we wanted to allow our algorithm to also select less than the maximum number of matrices as a feasible solution. Then, for each initial solution we generated random matrices with entries 0 and 1, whilst also making sure that those matrices were row concave. This is done row by row. We generate two random numbers and take the minimum as integer1 and the maximum as integer2. These numbers indicate the last index of the zeroes and ones. So in the array, index 1 till integer1 are zeros, integer1 + 1 till integer2 are ones and integer2 + 1 until the end are zeros. We also generated random weights for these matrices, keeping into account that the total weight does not exceed the value of W_{max}. For easy storage, we multiplied the matrices with the corresponding weights and stored these into a cell array.

1.3 Mating pool and parent selection

After generating the initial population, we start the selection of the mating pool and the parents. Both of these are done by a tournament selection in our code. We chose to do this because we found that using tournament selection gave a good balance between exploration and exploitation. Moreover, it is easier to either focus on exploration or exploitation by changing the tournament sizes, simply by changing their parameter values.

Firstly, the algorithm uses a function called `mating_pool_creator`, which creates a mating pool of size *mating* (one of the parameter values). The function generates l_1 distinct random integers between 1 and P, which will correspond to the indices of the input solutions. From these solutions, it then finds the solution with the lowest objective value and adds this to the mating pool. After doing so, it sets the objective value of this particular solution to $+\infty$. This makes sure that we don't select this solution again when we create the remainder of the mating pool. The function keeps on doing these until it reaches a size of *mating*.

After we have found the mating pool, we move to parent selection using the `parents_selection` function. It selects parent couples from the mating pool and stores these pairs in a matrix. Again, we first choose l_2 distinct random integers, now between 1 and *mating*, which correspond to the indices of the parents. From this group, we select the parent with the lowest objective value, and then we set the value of this parent to $+\infty$, to make sure we don't select the same integer for the second parent or for any next couple. We generate a new group of l_2 distinct random integers, from which we select the second parent. Together with the first parent, we add this as a couple to the matrix `parents`. This will be done until we reach rounded $P/2$ couple (because each couple will create 2 offspring solutions, we only need $P/2$ parent couples). Of course we need to take into account the fact that at some point, we may select l_2 solutions in the tournament of which the values have all been set to infinity. If this happens, we will reset all the values of the entire mating pool. If we had not done this, the algorithm would have selected the first solution in the mating pool, because they would all have a value of infinity.

1.4 Crossover and mutation

Crossover and mutation happen in our code in the `offspring_creation` function. We take as most important input variables the parent couples found during the parent selection. First, we need to check how many matrices each parent has. We decided to give the offspring the maximum number of matrices of both parents. For instance, if our first parent has 3 matrices and our second parent has 5 matrices, we will have the offspring have 5 matrices. To be able to generate matrices as soon as we run out of the 3 matrices for the first parent, we jump back to the first matrix. Hence, matrix 4 of the second parent will then be combined with matrix 1 of the first parent, and matrix 5 of the second parent with matrix 2 of the first parent. This is more clearly demonstrated in figure 1 below.

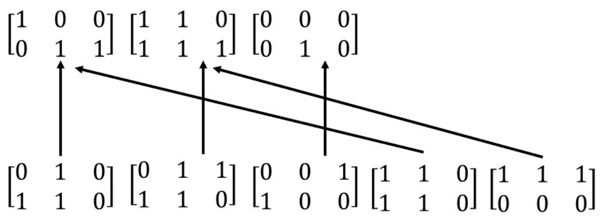


Figure 1: Combining parent matrices

To create the crossover, we extract the parent weights and matrices from the cell array we stored it in. After doing so, we combine the parent matrices by swapping half of the rows within these matrices. We chose for half the rows because in that case, the offspring would resemble both parents equally well. This results in two variations, which will correspond to the two children. We made the choice to only swap rows because this would ensure that the new solution would remain feasible without having to do too many intensive programming and calculations, which kept run-time lower.

To determine the weight, we just take the weight of the parent, i.e. child 1 will get the weights of parent 1 and child 2 the weights of parent 2. If parent 1 runs out of matrices (because parent 2 and therefore the offspring might have more matrices), we assume a weight of 0 for this parent for the remaining matrices. Again, we chose to do this because this would ensure that the total weights of the matrix stays below W_{max}, again making sure that the new solutions remain feasible.

For the mutation we decided to swap two random rows within each child. We chose to do this because it proved itself to be faster in the code, which means we could get to a better objective value in less time. Mutating the weights proved itself to be more complicated, which resulted in higher computation time and a less bigger overall effect. A mutation will occur with a set probability. We let the mutation occur in both children for the same rows. We chose for this because the crossover will already result in a big enough variation between the children when we look at the rows (if the amount of rows is bigger than 2). For example, row 3 and 8 will most likely not be the same for child 1 as for child 2, and swapping row 3 and 8 will therefore result in a different mutation for both children.

1.5 Choice of parameter values

In our algorithm, we have to make a choice of value for the following parameters:

- P: Population size, i.e. the number of (initial) solutions

- l_1: Size of the mating pool tournament
- *mating*: Size of the mating pool
- l_2: Size of the parent selection tournament
- $p_{mutation}$: Mutation probability

The size of P is very important for the speed of the algorithm. This makes sense, because if the population is very large, the algorithm has to generate a lot of new solutions at every iteration. Especially when working with large matrices, a high population will lead to a very long run-time. Therefore we chose to let the population size P depend on the size of the input matrix A. If A is an $m \times n$ matrix, this looks as follows:

$$P = max(200, round(2000 - \frac{1}{5} * m * n)) \tag{1}$$

We found by experimentation that having P be higher than 2000 didn't have much of an effect on the objective value. We let P be decreasing in the number of elements of A but with a minimal value of 200, so that even for very large matrices the algorithm will still select a sufficiently large population. We chose for a decreasing relationship between P and $m \times n$, because for larger problems, the computation time would become very large if we had kept the population size the same (or perhaps increase it).

For the value of l_1, we decided to let this be dependent on the population we found. In particular, we let $l_1 = 0.5P$. A high value of l_1 means that the algorithm focuses more on exploitation, because when the tournament is very large, it will likely capture the solutions with the best objective values and you then select the solution with the best objective value to join the mating group. However, a small value of l_1 will result in more exploration, because a small tournament leads to more selection of solutions with less optimal objective values. We therefore chose to take half of the population as size for the tournament. In our opinion this is the perfect balance between exploration and exploitation. Changing 0.5 to a higher number (below 1) would lead to more exploitation, whereas lowering 0.5 to being closer to 0 would lead to more exploration.

For the size of the mating pool, we let it depend on the size of the population. We chose a fraction of $\frac{1}{3}$, because this appeared as a good fraction by experimentation. Hence, we have that $mating = \frac{1}{3}P$. Another argument for keeping the mating pool 'not too large' is to avoid very long computation times for the parents selection.

Moving on to the size of the parent selection pool l_2, we take a fraction of the size of the mating pool *mating*. We found by experimentation that having a small value of l_2 resulted in better results for the final objective value. Therefore we let $l_2 = 0.1 * mating$. Because we already balanced exploration and exploitation in the selection of the mating pool, we decided to focus more on exploration for choosing the parents. A low value of l_2 does exactly that, which is why we decided on 0.1 as the multiplication factor.

Finally we have a mutation probability to apply random mutations to the offspring. We chose for a value of 0.2 for this probability, because by experimentation we found that this number applies just the right amount of variation to our solutions. A value smaller than 0.2 had barely any effect on the solutions and a value (much) larger than 0.2 altered the solution so much that the objective value would only keep increasing.

1.6 Fitness calculation

Since we store our matrices $B(i)$ in a cell, we did not come up with a smart way to immediately compute the fitness values from the cell entries. Instead, each time that we generate a new solution, we also keep track of the total of all the generated matrices. In other words, each time we generate new offspring, we multiply the row concave matrices with their weight and sum them up to the total of the solution. This way, each time a new solution is generated, we immediately have the total. We can then calculate the residual matrix S by subtracting the matrix M to be decomposed from the found total. After this we apply a double sum to this residual matrix, where all entries are first element-wise squared.

1.7 Stopping criteria

In our algorithm, we make use of 3 useful stopping criteria. The first one is change in objective value. If the global optimum does not change within `sameit` iterations, the algorithm stops. A second and very trivial stopping criterion is when the best found objective value in an iteration is 0. This means that an exact row concave decomposition is possible and that there is no need anymore to minimize the remainder matrix S. Our final stopping criterion is time. The algorithm keeps track of the amount of total time spent on computations and when the total time exceeds `T`, the algorithm also stops. Note that we also included the parameter *maxit* for pre-allocation purposes, but we set this parameter value to be sufficiently high so that it would not act as a stopping criterion.

2 Results

The following tables display the requested results from the assignment. We found that the seed we used had a big impact on the objective value we found. Therefore we ran our code 5 times for each test instance, without a set seed. This ensures that we account for the randomness in the solution. In the tables below, we report the average, maximum and minimum objective value of the 5 runs.

	$K_{max} = 8, W_{max} = 100$	$K_{max} = 12, W_{max} = 150$	$K_{max} = 16, W_{max} = 200$
10s	2,075,441	2,419,651	2,694,575
60s	1,475,817	1,495,933	1,678,282
120s	1,451,037	1,336,348	1,594,614
300s	1,468,353	1,234,090	1,343,365

Table 1: Minimum results for different values of K_{max}, W_{max} and T

	$K_{max} = 8, W_{max} = 100$	$K_{max} = 12, W_{max} = 150$	$K_{max} = 16, W_{max} = 200$
10s	2,098,848	2,487,362	2,864,494
60s	1,503,260	1,517,881	1,747,597
120s	1,498,323	1,391,315	1,759,865
300s	1,487,781	1,312,978	1,670,050

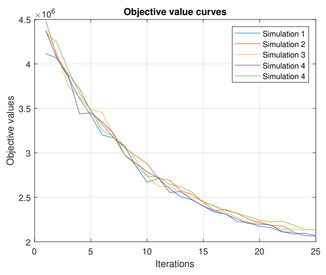
Table 2: Average results for different values of K_{max}, W_{max} and T

	$K_{max} = 8, W_{max} = 100$	$K_{max} = 12, W_{max} = 150$	$K_{max} = 16, W_{max} = 200$
10s	2,156,896	2,548,466	3,046,560
60s	1,527,036	1,544,381	1,818,556
120s	1,566,441	1,488,175	1,973,915
300s	1,514,384	1,387,681	2,113,617

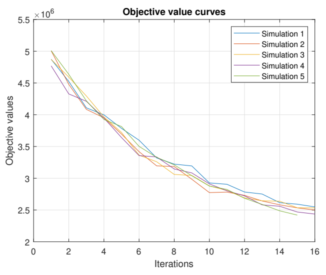
Table 3: Maximum results for different values of K_{max}, W_{max} and T

In the figures 2 and 3 we display the curves of the minimal objective value of each iteration for $T = 10$ and $T = 300$, for the different test instances. Based on the above tables and graphs below, we can make a few observations. The best found solution is found for the parameters $K_{max} = 12$ and $W_{max} = 150$ using a total time of 300 seconds. That the total time leading to the best solutions is 300 seconds was to be expected, because a larger total time provides more time to find a better solution. We did expect that we would find lower objective values for $K_{max} = 16$ and $W_{max} = 200$ opposed to $K_{max} = 12$ and $W_{max} = 150$, since the first would allow for more possibilities. However, this did not seem to be the case and we suspect that this is due to the fact that we did a lot of testing with $K_{max} = 12$ and $W_{max} = 150$. Therefore we mainly based our formulas for $P, l, mating$ and l_2 on the above testing. We suspect that the functions we came up with to determine the parameter values best suit the values $K_{max} = 12$

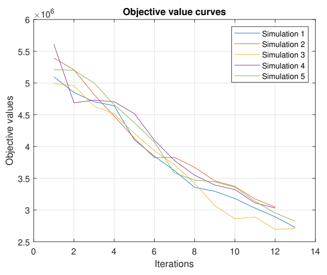
and $W_{max} = 150$, and that the fit might be worse for the different values of K_{max} and W_{max}. Choosing the parameters separately for each instance may lead to more optimal objective values, but this was not possible due to the nature of the function we had to write.



(a) $K = 8, W = 100$

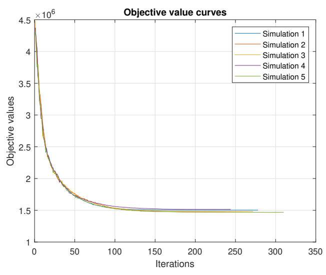


(b) $K = 12, W = 150$

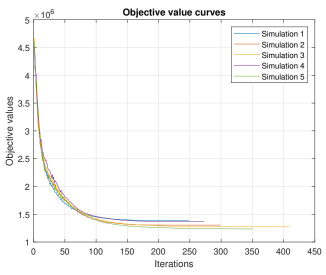


(c) $K = 16, W = 200$

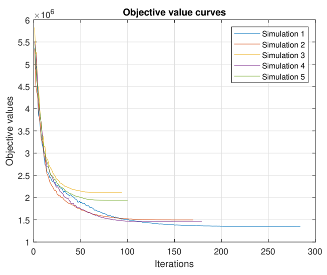
Figure 2: Simulations for T = 10



(a) $K = 8, W = 100$



(b) $K = 12, W = 150$



(c) $K = 16, W = 200$

Figure 3: Simulations for T = 300

One notable thing is also that the curves for the test instances with a higher K_{max} and W_{max} appear much more wiggly. Again, this might be due to the nature of our parameter value functions. However, this also could be caused by the fact that for a higher K_{max} and W_{max}, we will have a higher variation in the initial solutions because the algorithm is free the select from a broader range of matrices and weights. Performing crossover and mutations might therefore have a more drastic effect than for the instances with a lower K_{max} and W_{max}.

Another thing we noticed is the variance in the simulations when K_{max} and W_{max} increase. We clearly see when we go "from left to right" in the above graphs, that the lines become less overlapping. This is probably due to the fact that the algorithm gets more choices for the number of matrices and the total weight. This will probably lead to the algorithm sometimes making bad choices and getting stuck in a higher objective value.

We also see that, disregarding a few exceptions, the objective value becomes smaller over time. This suggests that time is an important stopping criterion. We see that our code is always above 2,000,000 after only 10 seconds, but is already much lower after 60 seconds. Our algorithm thus quickly finds improvements, but will converge slower and slower for more run time.

We would once again like to stress that our algorithm performed well for some particular seeds we tested during the creation of the algorithm. In the tables above we find a minimal objective value in the instance with $K_{max} = 12$ and $W_{max} = 150$, which was approximately 1.234 million. The best value we reached during testing was approximately 1.219 million, which is a small improvement. The fact that the algorithm doesn't perform quite as well for all randomness seeds is of course one of its shortcomings. Further ideas and implementations might cause the algorithm to perform better for all seeds, though testing this might prove itself to be a bit complicated due to randomness occurring.